Mysql 性能优化教程

目录

目录 1
背景及目标....2
Mysql 执行优化 2
认识数据索引....2
为什么使用数据索引能提高效率....2
如何理解数据索引的结构....2
优化实战范例....3
认识影响结果集....4
影响结果集的获取....4
影响结果集的解读....4
常见案例及优化思路....5
理解执行状态....7
常见关注重点....7
执行状态分析....8
分析流程....9
常见案例解析....11
总结....12
Mysql 运维优化 14
存储引擎类型....14
内存使用考量....14
性能与安全性考量....14
存储/写入压力优化....15
运维监控体系....15
Mysql 架构优化 17
架构优化目标....17
防止单点隐患....17
方便系统扩容....17
安全可控，成本可控....17
分布式方案....18
分库&拆表方案....18
反范式设计（冗余结构设计）....20
主从架构....21
故障转移处理....22
缓存方案....22
缓存结合数据库的读取....22
缓存结合数据库的写入....23
总结 24

VISIT…

背景及目标

- 厦门游家公司（4399.com）用于员工培训和分享。
- 针对用户群为已经使用过 mysql 环境，并有一定开发经验的工程师
- 针对高并发，海量数据的互联网环境。
- 本文语言为口语，非学术标准用语。
- 以实战和解决具体问题为主要目标，非应试，非常规教育。友情提醒，在校生学习本教程可能对成绩提高有害无益。
- 非技术挑战，非高端架构师培训，请高手自动忽略。
- 本文档在 2011 年 7 月-12 月持续更新，加强了影响结果集分析的内容并增补优化实战案例若干。

Mysql 执行优化

认识数据索引

为什么使用数据索引能提高效率

- 关系型数据库的数据索引（Btree 及常见索引结构）的存储是有序的。
- 在有序的情况下，通过索引查询一个数据是无需遍历索引记录的
- 关系型数据库数据索引的查询效率趋近于二分法查询效率，趋近于 log2(N)。
- 极端情况下（更新请求少，更新实时要求低，查询请求频繁），建立单向有序序列可替代数据索引。
- HASH 索引的查询效率是寻址操作，趋近于 1 次查询，比有序索引查询效率更高，但是不支持比对查询，区间查询，排序等操作，仅支持 key-value 类型查询。不是本文重点。

如何理解数据索引的结构

- 数据索引通常默认采用 btree 索引，（内存表也使用了 hash 索引）。
- 仅就有序前提而言，单向有序排序序列是查找效率最高的（二分查找，或者说折半查找），使用树形索引的目的是为了达到快速的更新和增删操作。
- 在极端情况下（比如数据查询需求量非常大，而数据更新需求极少，实时性要求不高，数据规模有限），直接使用单一排序序列，折半查找速度最快。

- 在进行索引分析和 SQL 优化时，可以将数据索引字段想象为单一有序序列，并以此作为分析的基础。涉及到复合索引情况，复合索引按照索引顺序拼凑成一个字段，想象为单一有序序列，并以此作为分析的基础。
- 一条数据查询只能使用一个索引，索引可以是多个字段合并的复合索引。但是一条数据查询不能使用多个索引。

优化实战范例

- 实战范例 1： ip 地址反查
  - 资源： Ip 地址对应表，源数据格式为 startip, endip, area
    源数据条数为 10 万条左右，呈很大的分散性
  - 目标： 需要通过任意 ip 查询该 ip 所属地区
    性能要求达到每秒 1000 次以上的查询效率
  - 挑战： 如使用 between startip and endip 这样的条件数据库操作，因为涉及两个字段的 between and, 无法有效使用索引。
    如果每次查询请求需要遍历 10 万条记录，根本不行。
  - 方法： 一次性排序（只在数据准备中进行，数据可存储在内存序列）
    折半查找（每次请求以折半查找方式进行）
- 实战范例 2：目标：查找与访问者同一地区的异性，按照最后登录时间逆序
  - 挑战：高访问量社区的高频查询，如何优化。
    查询 SQL: select * from user where area='$area' and sex='$sex' order by lastlogin desc limit 0,30;

  建立复合索引并不难， area+sex+lastlogin 三个字段的复合索引,如何理解？
  - 解读：首先，忘掉 btree，将索引字段理解为一个排序序列。

  另外，牢记数据查询只能使用一个索引，每个字段建立独立索引的情况下，也只能有一条索引被使用！

  如果只使用 area 会怎样？搜索会把符合 area 的结果全部找出来，然后在这里面遍历，选择命中 sex 的并排序。 遍历所有 area='$area'数据！

  如果使用了 area+sex，略好，仍然要遍历所有 area='$area' and sex='$sex'数据，然后在这个基础上排序!!

  Area+sex+lastlogin 复合索引时（切记 lastlogin 在最后），该索引基于 area+sex+lastlogin 三个字段合并的结果排序，该列表可以想象如下。

  广州女$时间 1
  广州女$时间 2
  广州女$时间 3
  …
  广州男
  ….
  深圳女
  ….

  数据库很容易命中到 area+sex 的边界，并且基于下边界向上追溯 30 条记录，搞定！在索引中迅速命中所有结果，无需二次遍历！

认识影响结果集

影响结果集的获取

- 通过 Explain 分析 SQL，查看 rows 列内容
- 通过慢查询日志的 Rows_examined: 后面的数字
- 影响结果集数字是查询优化的重要中间数字，工程师在开发和调试过程中，应随时关注这一数字。

影响结果集的解读

- 查询条件与索引的关系决定影响结果集。
  - 影响结果集不是输出结果数，不是查询返回的记录数，而是索引所扫描的结果数。
  - 范例 select * from user where area='厦门' and sex='女'
    - 假设 索引为 area
    - 假设 User 表中 area='厦门'的有 125000 条，而搜索返回结果为 60233 条。
    - 影响结果集是 125000 条，索引先命中 125000 条厦门用户，再遍历以 sex='女'进行筛选操作，得到 60233 条结果。
    - 如果该 SQL 增加 limit 0,30 的后缀。查询时，先命中 area='厦门'，然后依顺序执行 sex='女' 筛选操作，直到满足可以返回 30 条为止，所涉及记录数未知。除非满足条件的结果不足 30 条，否则不会遍历 125000 条记录。
    - 但是如果 SQL 中涉及了排序操作，比如 order by lastlogin desc 再有 limit 0,30 时，排序需要遍历所有 area='厦门' 的记录，而不是满足即止。
- 影响结果集越趋近于实际输出或操作的目标结果集，索引效率越高。
- 影响结果集与查询开销的关系可以理解为线性相关。减少一半影响结果集，即可提升一倍查询效率！当一条搜索 query 可以符合多个索引时，选择影响结果集最少的索引。
- SQL 的优化，核心就是对结果集的优化，认识索引是增强对结果集的判断，基于索引的认识，可以在编写 SQL 的时候，对该 SQL 可能的影响结果集有预判，并做出适当的优化和调整。
- Limit 的影响，需要斟酌对待
  - 如果索引与查询条件和排序条件完全命中，影响结果集就是 limit 后面的数字（$start + $end），比如 limit 200,30 影响结果集是 230. 而不是 30.
  - 如果索引只命中部分查询条件，甚至无命中条件，在无排序条件情况下，会在索引命中的结果集 中遍历到满足所有其他条件为止。比如 select * from user limit 10; 虽然没用到索引，但是因为不涉及二次筛选和排序，系统直接返回前 10 条结果，影响结果集依然只有 10 条，就不存在效率影响。
  - 如果搜索所包含的排序条件没有被索引命中，则系统会遍历是所有索引所命中的结果，并且排序。例如 Select * from user order by timeline desc limit 10; 如果 timeline 不是索引，影响结果集是全表，就存在需要全表数据排序，这个效率影响就巨大。再比如 Select * from user where area='厦门' order by timeline

desc limit 10; 如果 area 是索引，而 area+timeline 未建立索引，则影响结果集是所有命中 area='厦门'的用户，然后在影响结果集内排序。

常见案例及优化思路

- 毫秒级优化案例
  - 某游戏用户进入后显示最新动态，SQL 为 select * from userfeed where uid=$uid order by timeline desc limit 20; 主键为$uid 。 该 SQL 每天执行数百万次之多，高峰时数据库负载较高。 通过 show processlist 显示大量进程处于 Sending data 状态。没有慢查询记录。 仔细分析发现，因存在较多高频用户访问，命中 uid=$uid 的影响结果集通常在几百到几千，在上千条影响结果集情况下，该 SQL 查询开销通常在 0.01 秒左右。 建立 uid+timeline 复合索引，将排序引入到索引结构中，影响结果集就只有 limit 后面的数字，该 SQL 查询开销锐减至 0.001 秒，数据库负载骤降。
- Innodb 锁表案例
  - 某游戏数据库使用了 innodb，innodb 是行级锁，理论上很少存在锁表情况。出现了一个 SQL 语句(delete from tabname where xid=…)，这个 SQL 非常用 SQL，仅在特定情况下出现，每天出现频繁度不高（一天仅 10 次左右），数据表容量百万级，但是这个 xid 未建立索引，于是悲惨的事情发生了，当执行这条 delete 的时候，真正删除的记录非常少，也许一到两条，也许一条都没有；但是！由于这个 xid 未建立索引，delete 操作时遍历全表记录，全表被 delete 操作锁定，select 操作全部被 locked，由于百万条记录遍历时间较长，期间大量 select 被阻塞，数据库连接过多崩溃。

    这种非高发请求，操作目标很少的 SQL，因未使用索引，连带导致整个数据库的查询阻塞，需要极大提高警觉。
- 实时排名策略优化
  - 背景： 用户提交游戏积分，显示实时排名。
  - 原方案：
    - 提交积分是插入记录，略，
    - select count(*) from jifen where gameid=$gameid and fenshu>$fenshu
  - 问题与挑战
    - 即便索引是 gameid+fenshu 复合索引，涉及 count 操作，当分数较低时，影响结果集巨大，查询效率缓慢，高峰期会导致连接过多。
  - 优化思路
    - 减少影响结果集，又要取得实时数据，单纯从 SQL 上考虑，不太有方法。
    - 将游戏积分预定义分成数个积分断点，然后分成积分区间，原始状态，每个区间设置一个统计数字项，初始为 0。
    - 每次积分提交时，先确定该分数属于哪两个区间之间，这个操作非常简单，因为区间是预定义的，而且数量很少，只需遍历即可，找到最该分数符合的区间， 该区间的统计数字项（独立字段，可用内存处理，异步回写数据库或文件）+1。 记录该区间上边界数字为$duandian。
    - SQL: select count(*) from jifen where gameid=$gameid and fenshu>$fenshu and fenshu<$duandian，如果处于第一区间，则无需

$duandian，这样因为第一区间本身也是最好的成绩，影响结果集不会很多。 通过该 SQL 获得其在该区间的名次。

- 获取前面区间的总数总和。（该数字是直接从上述提到的区间统计数字获取，不需要进行 count 操作）将区间内名次+前区间的统计数字和，获得总名次。
- 该方法关键在于，积分区间需要合理定义，保证积分提交成绩能平均散落在不同区间。
- 如涉及较多其他条件，如日排行，总排行，以及其他独立用户去重等，请按照影响结果集思路自行发挥。

◆ Redis 方案

- Redis 数据结构包括 String, list, dict 和 Zset 四种，在本案例中是非常好的替代数据库的方案，本文档只做简介，不做额外扩展。
- String 哈希索引，key-value 结构，主键查询效率极高，不支持排序，比较查询。
- List 队列结构，在数据异步写入处理中可以替代 memcache。
- Dict 数组结构，存储结构化，序列化内容，可以针对数组中的特定列进行操作。
- Zset 有序数组结构，分两个子结构，第一是多层树形的存储结构，第二是每个树形节点的计数器，这样类似于前面的分段方式，可以理解为多层分段方式，所以查询效率更高，缺点是更新效率有所增加。

■ 论坛翻页优化

◆ 背景，常见论坛帖子页 SQL: select * from post where tagid=$tagid order by lastpost limit $start, $end 翻页 。索引为 tagid+lastpost 复合索引

◆ 挑战， 超级热帖，几万回帖，用户频频翻到末页，limit 25770,30 一个操作下来，影响结果集巨大(25770+30)，查询缓慢。

◆ 解决方法：

- 只涉及上下翻页情况
  - 每次查询的时候将该页查询结果中最大的 $lastpost 和最小的分别记录为 $minlastpost 和 $maxlastpost ，上翻页查询为 select * from post where tagid=$tagid and lastpost<$minlastpost order by lastpost desc limit 30; 下翻页为 select * from post where tagid=$tagid and lastpost>$maxlastpost order by lastpost limit 30; 使用这种方式，影响结果集只有 30 条，效率极大提升。
- 涉及跳转到任意页
  - 互联网上常见的一个优化方案可以这样表述，select * from post where tagid=$tagid and lastpost>=(select lastpost from post where tagid=$tagid order by lastpost limit $start,1) order by lastpost limit 30; 或者 select * from post where pid in (select pid from post where tagid=$tagid order by lastpost limit $start,30); (第 2 条 S 语法在新的 mysql 版本已经不支持，新版本 mysql in 的子语句不再支持 limit 条件，但可以分解为两条 SQL 实现，原理不变，不做赘述)
  - 以上思路在于，子查询的影响结果集仍然是$start +30，但是数据获取的过程（Sending data 状态）发生在索引文件中，而不是数据表文件，这样所需要的系统开销就比前一种普通的查询低一个数量级，而主查

询的影响结果集只有 30 条，几乎无开销。但是切记，这里仍然涉及了太多的影响结果集操作。

- 延伸问题：
  - 来自于 uchome 典型查询 SELECT * FROM uchome_thread WHERE tagid='73820' ORDER BY displayorder DESC, lastpost DESC LIMIT $start,30;
  - 如果换用 如上方法，上翻页代码 SELECT * FROM uchome_thread WHERE tagid='73820' and lastpost<$minlastpost ORDER BY displayorder DESC,lastpost DESC LIMIT 0,30; 下翻页代码 SELECT * FROM uchome_thread WHERE tagid='73820' and lastpost>$maxlastpost ORDER BY displayorder DESC, lastpost ASC LIMIT 0,30;
  - 这里涉及一个 order by 索引可用性问题，当 order by 中 复合索引的字段，一个是 ASC，一个是 DESC 时，其排序无法在索引中完成。 所以只有上翻页可以正确使用索引，影响结果集为 30。下翻页无法在排序中正确使用索引，会命中所有索引内容然后排序，效率低下。

- 总结：
  - 基于影响结果集的理解去优化，不论从数据结构，代码，还是涉及产品策略上，都需要贯彻下去。
  - 涉及 limit $start,$num 的搜索，如果$start 巨大，则影响结果集巨大，搜索效率会非常难过低，尽量用其他方式改写为 limit 0,$num； 确系无法改写的情况下，先从索引结构中获得 limit $start,$num 或 limit $start,1 ；再用 in 操作或基于索引序的 limit 0,$num 二次搜索。
  - 请注意，我这里永远不会讲关于外键和 join 的优化，因为在我们的体系里，这是根本不允许的！ 架构优化部分会解释为什么。

理解执行状态

常见关注重点

- 慢查询日志，关注重点如下
  - 是否锁定，及锁定时间
    - 如存在锁定，则该慢查询通常是因锁定因素导致，本身无需优化，需解决锁定问题。
  - 影响结果集
    - 如影响结果集较大，显然是索引项命中存在问题，需要认真对待。
- Explain 操作
  - 索引项使用
    - 不建议用 using index 做强制索引，如未如预期使用索引，建议重新斟酌表结构和索引设置。
  - 影响结果集
    - 这里显示的数字不一定准确，结合之前提到对数据索引的理解来看，还记得嘛？就把索引当作有序序列来理解，反思 SQL。
- Set profiling , show profiles for query 操作

- 执行开销
  - 注意，有问题的 SQL 如果重复执行，可能在缓存里，这时要注意避免缓存影响。通过这里可以看到。
  - 执行时间超过 0.005 秒的频繁操作 SQL 建议都分析一下。
  - 深入理解数据库执行的过程和开销的分布
- Show processlist 执行状态监控
  - 这是在数据库负载波动时经常进行的一项操作
  - 具体参见如下

执行状态分析

- Sleep 状态
  - 通常代表资源未释放，如果是通过连接池，sleep 状态应该恒定在一定数量范围内
  - 实战范例： 因前端数据输出时（特别是输出到用户终端）未及时关闭数据库连接，导致因网络连接速度产生大量 sleep 连接，在网速出现异常时，数据库 too many connections 挂死。
  - 简单解读，数据查询和执行通常只需要不到 0.01 秒，而网络输出通常需要 1 秒左右甚至更长，原本数据连接在 0.01 秒即可释放，但是因为前端程序未执行 close 操作，直接输出结果，那么在结果未展现在用户桌面前，该数据库连接一直维持在 sleep 状态！
- Waiting for net, reading from net, writing to net
  - 偶尔出现无妨
  - 如大量出现，迅速检查数据库到前端的网络连接状态和流量
  - 案例: 因外挂程序，内网数据库大量读取，内网使用的百兆交换迅速爆满，导致大量连接阻塞在 waiting for net，数据库连接过多崩溃
- Locked 状态
  - 有更新操作锁定
  - 通常使用 innodb 可以很好的减少 locked 状态的产生，但是切记，更新操作要正确使用索引，即便是低频次更新操作也不能疏忽。如上影响结果集范例所示。
  - 在 myisam 的时代，locked 是很多高并发应用的噩梦。所以 mysql 官方也开始倾向于推荐 innodb。
- Copy to tmp table
  - 索引及现有结构无法涵盖查询条件，才会建立一个临时表来满足查询要求，产生巨大的恐怖的 i/o 压力。
  - 很可怕的搜索语句会导致这样的情况，如果是数据分析，或者半夜的周期数据清理任务，偶尔出现，可以允许。频繁出现务必优化之。
  - Copy to tmp table 通常与连表查询有关，建议逐渐习惯不使用连表查询。
  - 实战范例：
    - 某社区数据库阻塞，求救，经查，其服务器存在多个数据库应用和网站，其中一个不常用的小网站数据库产生了一个恐怖的 copy to tmp table 操作，导致整个硬盘 i/o 和 cpu 压力超载。Kill 掉该操作一切恢复。
- Sending data
  - Sending data 并不是发送数据，别被这个名字所欺骗，这是从物理磁盘获取数

据的进程，如果你的影响结果集较多，那么就需要从不同的磁盘碎片去抽取数据，

- 偶尔出现该状态连接无碍。
- 回到上面影响结果集的问题，一般而言，如果 sending data 连接过多，通常是某查询的影响结果集过大，也就是查询的索引项不够优化。
- 前文提到影响结果集对 SQL 查询效率线性相关，主要就是针对这个状态的系统开销。
- 如果出现大量相似的 SQL 语句出现在 show proesslist 列表中，并且都处于 sending data 状态，优化查询索引，记住用影响结果集的思路去思考。

- Storing result to query cache
  - 出现这种状态，如果频繁出现，使用 set profiling 分析，如果存在资源开销在 SQL 整体开销的比例过大（即便是非常小的开销，看比例），则说明 query cache 碎片较多
  - 使用 flush query cache 可即时清理，也可以做成定时任务
  - Query cache 参数可适当酌情设置。
- Freeing items
  - 理论上这玩意不会出现很多。偶尔出现无碍
  - 如果大量出现，内存，硬盘可能已经出现问题。比如硬盘满或损坏。
  - i/o 压力过大时，也可能出现 Free items 执行时间较长的情况。
- Sorting for …
  - 和 Sending data 类似，结果集过大，排序条件没有索引化，需要在内存里排序，甚至需要创建临时结构排序。
- 其他
  - 还有很多状态，遇到了，去查查资料。基本上我们遇到其他状态的阻塞较少，所以不关心。

分析流程

- 基本流程
  - 详细了解问题状况
    - Too many connections 是常见表象，有很多种原因。
    - 索引损坏的情况在 innodb 情况下很少出现。
    - 如出现其他情况应追溯日志和错误信息。
  - 了解基本负载状况和运营状况
    - 基本运营状况
      - 当前每秒读请求
      - 当前每秒写请求
      - 当前在线用户
      - 当前数据容量
    - 基本负载情况
      - 学会使用这些指令
        - Top
        - Vmstat
        - uptime

        - iostat
        - df
    - Cpu 负载构成
        - 特别关注 i/o 压力(wa%)
        - 多核负载分配
    - 内存占用
        - Swap 分区是否被侵占
        - 如 Swap 分区被侵占，物理内存是否较多空闲
    - 磁盘状态
        - 硬盘满和 inode 节点满的情况要迅速定位和迅速处理
- 了解具体连接状况
    - 当前连接数
        - Netstat –an|grep 3306|wc –l
        - Show processlist
    - 当前连接分布 show processlist
        - 前端应用请求数据库不要使用 root 帐号!
            - Root 帐号比其他普通帐号多一个连接数许可。
            - 前端使用普通帐号，在 too many connections 的时候 root 帐号仍可以登录数据库查询 show processlist!
            - 记住，前端应用程序不要设置一个不叫 root 的 root 帐号来糊弄!非 root 账户是骨子里的，而不是名义上的。
        - 状态分布
            - 不同状态代表不同的问题，有不同的优化目标。
            - 参见如上范例。
        - 雷同 SQL 的分布
            - 是否较多雷同 SQL 出现在同一状态
    - 当前是否有较多慢查询日志
        - 是否锁定
        - 影响结果集
- 频繁度分析
    - 写频繁度
        - 如果 i/o 压力高，优先分析写入频繁度
        - Mysqlbinlog 输出最新 binlog 文件，编写脚本拆分
        - 最多写入的数据表是哪个
        - 最多写入的数据 SQL 是什么
        - 是否存在基于同一主键的数据内容高频重复写入？
            - 涉及架构优化部分，参见架构优化-缓存异步更新
    - 读取频繁度
        - 如果 cpu 资源较高，而 i/o 压力不高，优先分析读取频繁度
        - 程序中在封装的 db 类增加抽样日志即可，抽样比例酌情考虑，以不显著影响系统负载压力为底线。
        - 最多读取的数据表是哪个
        - 最多读取的数据 SQL 是什么
            - 该 SQL 进行 explain 和 set profiling 判定

        - 注意判定时需要避免 query cache 影响
            - 比如，在这个 SQL 末尾增加一个条件子句 and 1=1 就可以避免从 query cache 中获取数据，而得到真实的执行状态分析。
    - 是否存在同一个查询短期内频繁出现的情况
        - 涉及前端缓存优化
- 抓大放小，解决显著问题
    - 不苛求解决所有优化问题，但是应以保证线上服务稳定可靠为目标。
    - 解决与评估要同时进行，新的策略或解决方案务必经过评估后上线。

常见案例解析

- 现象：服务器出现 too many connections 阻塞
    - 入手点：
        - 查看服务器状态，cpu 占用，内存占用，硬盘占用，硬盘 i/o 压力
        - 查看网络流量状态，mysql 与应用服务器的输入输出状况
        - 通过 Show processlist 查看当前运行清单
            - 注意事项，日常应用程序连接数据库不要使用 root 账户，保证故障时可以通过 root 进入数据库查看 show processlist。
    - 状态分析：
        - 参见如上执行状态清单，根据连接状态的分布去确定原因。
    - 紧急恢复
        - 在确定故障原因后，应通过 kill 掉阻塞进程的方式 立即恢复数据库。
    - 善后处理
        - 以下针对常见问题简单解读
        - Sleep 连接过多导致，应用端及时释放连接，排查关联因素。
        - Locked 连接过多，如源于 myisam 表级锁，更 innodb 引擎;如源于更新操作使用了不恰当的索引或未使用索引，改写更新操作 SQL 或建立恰当索引。
        - Sending data 连接过多，用影响结果集的思路优化 SQL 查询，优化表索引结构。
        - Free items 连接过多，i/o 压力过大 或硬盘故障
        - Waiting for net , writing to net 连接过多， mysql 与应用服务器连接阻塞。
        - 其他仍参见如上执行状态清单所示分析。
        - 如涉及不十分严格安全要求的数据内容，可用定期脚本跟踪请求进程，并 kill 掉僵死进程。如数据安全要求较严格，则不能如此进行。
- 现象：数据库负载过高，响应缓慢。
    - 入手点：
        - 查看 cpu 状态，服务器负载构成
    - 分支 1：i/o 占用过高。
        - 步骤 1： 检查内存是否占用 swap 分区，排除因内存不足导致的 i/o 开销。
        - 步骤 2：通过 iostat 指令分析 i/o 是否集中于数据库硬盘，是否是写入度较高。
        - 步骤 3：如果压力来自于写，使用 mysqlbinlog 解开最新的 binlog 文件。
        - 步骤 4：编写日志分析脚本或 grep 指令，分析每秒写入频度和写入内容。
            - 写入频度不高，则说明 i/o 压力另有原因或数据库配置不合理。
        - 步骤 5：编写日志分析脚本或 grep 指令，分析写入的数据表构成，和写入的

目标构成。

- 步骤 6：编写日志分析脚本，分析是否存在同一主键的重复写入。 比如出现大量 update post set views=views+1 where tagid=****的操作，假设在一段时间内出现了 2 万次，而其中不同的 tagid 有 1 万次，那么就是有 50%的请求是重复 update 请求，有可以通过异步更新合并的空间。
- 提示一下，以上所提及的日志分析脚本编写，正常情况下不应超过 1 个小时，而对系统负载分析所提供的数据支持价值是巨大的，对性能优化方案的选择是非常有意义的，如果您认为这项工作是繁冗而且复杂的工作，那么一定是在分析思路和目标把握上出现了偏差。

- 分支 2：i/o 占用不高，CPU 占用过高
  - 步骤 1：查看慢查询日志
  - 步骤 2：不断刷新查看 Show processlist 清单，并把握可能频繁出现的处于 Sending data 状态的 SQL。
  - 步骤 3：记录前端执行 SQL
    - 于前端应用程序执行查询的封装对象内，设置随机采样，记录前端执行的 SQL，保证有一定的样本规模，并且不会带来前端 i/o 负载的激增。
    - 基于采样率和记录频率，获得每秒读请求次数数据指标。
    - 编写日志分析脚本，分析采样的 SQL 构成，所操作的数据表，所操作的主键。
    - 对频繁重复读取的 SQL(完全一致的 SQL)进行判定，是否数据存在频繁变动，是否需要实时展现最新数据，如有可能，缓存化，并预估缓存命中率。
    - 对频繁读取但不重复的(SQL 结构一致，但条件中的数据不一致)SQL 进行判定，是否索引足够优化，影响结果集与输出结果是否足够接近。
  - 步骤 4：将导致慢查询的 SQL 或频繁出现于 show processlist 状态的 SQL，或采样记录的频繁度 SQL 进行分析，按照影响结果集的思路和索引理解来优化。
  - 步骤 5：对如上难以界定问题的 SQL 进行 set profiling 分析。
  - 步骤 6：优化后分析继续采样跟踪分析。并跟踪比对结果。
- 善后处理
  - 日常跟踪脚本，不断记录一些状态信息。保证每个时间节点都能回溯。
  - 确保随时能了解服务器的请求频次，读写请求的分布。
  - 记录一些未造成致命影响的隐患点，可暂不解决，但需要记录。
  - 如确系服务器请求频次过高，可基于负载分布决定硬件扩容方案，比如 i/o 压力过高可考虑固态硬盘；内存占用 swap 可考虑增加内容容量等。用尽可能少的投入实现最好的负载支撑能力，而不是简单的买更多服务器。

总结

- 要学会怎样分析问题，而不是单纯拍脑袋优化
- 慢查询只是最基础的东西，要学会优化 0.01 秒的查询请求。
- 当发生连接阻塞时，不同状态的阻塞有不同的原因，要找到原因，如果不对症下药，就会南辕北辙
  - 范例：如果本身系统内存已经超载，已经使用到了 swap，而还在考虑加大缓存来优化查询，那就是自寻死路了。
- 影响结果集是非常重要的中间数据和优化指标，学会理解这一概念，理论上影响结

果集与查询效率呈现非常紧密的线性相关。

- 监测与跟踪要经常做，而不是出问题才做
  - 读取频繁度抽样监测
    - 全监测不要搞，i/o 吓死人。
    - 按照一个抽样比例抽样即可。
    - 针对抽样中发现的问题，可以按照特定 SQL 在特定时间内监测一段全查询记录，但仍要考虑 i/o 影响。
  - 写入频繁度监测
    - 基于 binlog 解开即可，可定时或不定时分析。
  - 微慢查询抽样监测
    - 高并发情况下，查询请求时间超过 0.01 秒甚至 0.005 秒的，建议酌情抽样记录。
  - 连接数预警监测
    - 连接数超过特定阈值的情况下，虽然数据库没有崩溃，建议记录相关连接状态。
- 学会通过数据和监控发现问题，分析问题，而后解决问题顺理成章。特别是要学会在日常监控中发现隐患，而不是问题爆发了才去处理和解决。

Mysql 运维优化

存储引擎类型

- Myisam 速度快，响应快。表级锁是致命问题。
- Innodb 目前主流存储引擎
  - 行级锁
    - 务必注意影响结果集的定义是什么
    - 行级锁会带来更新的额外开销，但是通常情况下是值得的。
  - 事务提交
    - 对 i/o 效率提升的考虑
    - 对安全性的考虑
- HEAP 内存引擎
  - 频繁更新和海量读取情况下仍会存在锁定状况

内存使用考量

- 理论上，内存越大，越多数据读取发生在内存，效率越高
- Query cache 的使用
  - 如果前端请求重复度不高，或者应用层已经充分缓存重复请求，query cache 不必设置很大，甚至可以不设置。
  - 如果前端请求重复度较高，无应用层缓存，query cache 是一个很好的偷懒选择
    - 对于中等以下规模数据库应用，偷懒不是一个坏选择。
    - 如果确认使用 query cache，记得定时清理碎片，flush query cache.
- 要考虑到现实的硬件资源和瓶颈分布
- 学会理解热点数据，并将热点数据尽可能内存化
  - 所谓热点数据，就是最多被访问的数据。
  - 通常数据库访问是不平均的，少数数据被频繁读写，而更多数据鲜有读写。
  - 学会制定不同的热点数据规则，并测算指标。
    - 热点数据规模，理论上，热点数据越少越好，这样可以更好的满足业务的增长趋势。
    - 响应满足度，对响应的满足率越高越好。
    - 比如依据最后更新时间，总访问量，回访次数等指标定义热点数据，并测算不同定义模式下的热点数据规模

性能与安全性考量

- 数据提交方式
  - innodb_flush_log_at_trx_commit = 1 每次自动提交，安全性高，i/o 压力大
  - innodb_flush_log_at_trx_commit = 2 每秒自动提交，安全性略有影响，i/o 承载

强。

- 日志同步
  - Sync-binlog =1 每条自动更新，安全性高，i/o 压力大
  - Sync-binlog = 0 根据缓存设置情况自动更新，存在丢失数据和同步延迟风险，i/o 承载力强。
  - 个人建议保存 binlog 日志文件，便于追溯 更新操作和系统恢复。
  - 如对日志文件的 i/o 压力有担心，在内存宽裕的情况下，可考虑将 binlog 写入到诸如 /dev/shm 这样的内存映射分区，并定时将旧有的 binlog 转移到物理硬盘。
- 性能与安全本身存在相悖的情况，需要在业务诉求层面决定取舍
  - 学会区分什么场合侧重性能，什么场合侧重安全
  - 学会将不同安全等级的数据库用不同策略管理

存储/写入压力优化

- 顺序读写性能远高于随机读写
- 将顺序写数据和随机读写数据分成不同的物理磁盘进行，有助于 i/o 压力的疏解
  - 数据库文件涉及索引等内容，写入是随即写
  - binlog 文件是顺序写
  - 淘宝数据库存储优化是这样处理的
- 部分安全要求不高的写入操作可以用 /dev/shm 分区存储，简单变成内存写。
- 多块物理硬盘做 raid10，可以提升写入能力
- 关键存储设备优化，善于比对不同存储介质的压力测试数据。
  - 例如 fusion-io 在新浪和淘宝都有较多使用。
- 涉及必须存储较为庞大的数据量时
  - 压缩存储，可以通过增加 cpu 开销（压缩算法）减少 i/o 压力。前提是你确认 cpu 相对空闲而 i/o 压力很大。 新浪微博就是压缩存储的典范。
  - 通过 md5 去重存储，案例是 QQ 的文件共享，以及 dropbox 这样的共享服务，如果你上传的是一个别人已有的文件，计算 md5 后，直接通过 md5 定位到原有文件，这样可以极大减少存储量。涉及文件共享，头像共享，相册等应用，通过这种方法可以减少超过 70%的存储规模，对硬件资源的节省是相当巨大的。缺点是，删除文件需要甄别该 md5 是否有其他人使用。 去重存储，用户量越多，上传文件越多，效率越高！
  - 文件尽量不要存储到数据库内。尽量使用独立的文件系统存储，该话题不展开。

运维监控体系

- 系统监控
  - 服务器资源监控
    - Cpu，内存，硬盘空间，i/o 压力
    - 设置阈值报警
  - 服务器流量监控
    - 外网流量，内网流量

        - 设置阈值报警
    - 连接状态监控
        - Show processlist 设置阈值，每分钟监测，超过阈值记录
- 应用监控
    - 慢查询监控
        - 慢查询日志
        - 如果存在多台数据库服务器，应有汇总查阅机制。
    - 请求错误监控
        - 高频繁应用中，会出现偶发性数据库连接错误或执行错误，将错误信息记录到日志，查看每日的比例变化。
        - 偶发性错误，如果数量极少，可以不用处理，但是需时常监控其趋势。
        - 会存在恶意输入内容，输入边界限定缺乏导致执行出错，需基于此防止恶意入侵探测行为。
    - 微慢查询监控
        - 高并发环境里，超过 0.01 秒的查询请求都应该关注一下。
    - 频繁度监控
        - 写操作，基于 binlog，定期分析。
        - 读操作，在前端 db 封装代码中增加抽样日志，并输出执行时间。
        - 分析请求频繁度是开发架构 进一步优化的基础
        - 最好的优化就是减少请求次数！
- 总结：
    - 监控与数据分析是一切优化的基础。
    - 没有运营数据监测就不要妄谈优化！
    - 监控要注意不要产生太多额外的负载，不要因监控带来太多额外系统开销

Mysql 架构优化

架构优化目标

防止单点隐患

- 所谓单点隐患，就是某台设备出现故障，会导致整体系统的不可用，这个设备就是单点隐患。
- 理解连带效应，所谓连带效应，就是一种问题会引发另一种故障，举例而言，memcache+mysql 是一种常见缓存组合，在前端压力很大时，如果 memcache 崩溃，理论上数据会通过 mysql 读取，不存在系统不可用情况，但是 mysql 无法对抗如此大的压力冲击，会因此连带崩溃。因 A 系统问题导致 B 系统崩溃的连带问题，在运维过程中会频繁出现。
  - 实战范例： 在 mysql 连接不及时释放的应用环境里，当网络环境异常（同机房友邻服务器遭受拒绝服务攻击，出口阻塞），网络延迟加剧，空连接数急剧增加，导致数据库连接过多崩溃。
  - 实战范例 2：前端代码 通常我们封装 mysql_connect 和 memcache_connect，二者的顺序不同，会产生不同的连带效应。如果 mysql_connect 在前，那么一旦 memcache 连接阻塞，会连带 mysql 空连接过多崩溃。
  - 连带效应是常见的系统崩溃，日常分析崩溃原因的时候需要认真考虑连带效应的影响，头疼医头，脚疼医脚是不行的。

方便系统扩容

- 数据容量增加后，要考虑能够将数据分布到不同的服务器上。
- 请求压力增加时，要考虑将请求压力分布到不同服务器上。
- 扩容设计时需要考虑防止单点隐患。

安全可控，成本可控

- 数据安全，业务安全
- 人力资源成本>带宽流量成本>硬件成本
  - 成本与流量的关系曲线应低于线性增长（流量为横轴，成本为纵轴）。
  - 规模优势
- 本教程仅就与数据库有关部分讨论，与数据库无关部门请自行参阅其他学习资料。

分布式方案

分库&拆表方案

- 基本认识
  - 用分库&拆表是解决数据库容量问题的唯一途径。
  - 分库&拆表也是解决性能压力的最优选择。
  - 分库 – 不同的数据表放到不同的数据库服务器中（也可能是虚拟服务器）
  - 拆表 – 一张数据表拆成多张数据表，可能位于同一台服务器，也可能位于多台服务器（含虚拟服务器）。
- 去关联化原则
  - 摘除数据表之间的关联，是分库的基础工作。
  - 摘除关联的目的是，当数据表分布到不同服务器时，查询请求容易分发和处理。
  - 学会理解反范式数据结构设计，所谓反范式，第一要点是不用外键，不允许 Join 操作，不允许任何需要跨越两个表的查询请求。第二要点是适度冗余减少查询请求，比如说，信息表，fromuid, touid, message 字段外，还需要一个 fromuname 字段记录用户名，这样查询者通过 touid 查询后，能够立即得到发信人的用户名，而无需进行另一个数据表的查询。
  - 去关联化处理会带来额外的考虑，比如说，某一个数据表内容的修改，对另一个数据表的影响。这一点需要在程序或其他途径去考虑。
- 分库方案
  - 安全性拆分
    - 将高安全性数据与低安全性数据分库，这样的好处第一是便于维护，第二是高安全性数据的数据库参数配置可以以安全优先，而低安全性数据的参数配置以性能优先。参见运维优化相关部分。
  - 基于业务逻辑拆分
    - 根据数据表的内容构成，业务逻辑拆分，便于日常维护和前端调用。
    - 基于业务逻辑拆分，可以减少前端应用请求发送到不同数据库服务器的频次，从而减少链接开销。
    - 基于业务逻辑拆分，可保留部分数据关联，前端 web 工程师可在限度范围内执行关联查询。
  - 基于负载压力拆分
    - 基于负载压力对数据结构拆分，便于直接将负载分担给不同的服务器。
    - 基于负载压力拆分，可能拆分后的数据库包含不同业务类型的数据表，日常维护会有一定的烦恼。
  - 混合拆分组合
    - 基于安全与业务拆分为数据库实例，但是可以使用不同端口放在同一个服务器上。
    - 基于负载可以拆分为更多数据库实例分布在不同数据库上
    - 例如，
      - 基于安全拆分出 A 数据库实例，
      - 基于业务拆分出 B,C 数据库实例，
      - C 数据库存在较高负载，基于负载拆分为 C1,C2,C3,C4 等 实例。

- 数据库服务器完全可以做到 A+B+C1 为一台，C2,C3,C4 各单独一台。

- 分表方案
  - 数据量过大或者访问压力过大的数据表需要切分
  - 纵向分表（常见为忙闲分表）
    - 单数据表字段过多，可将频繁更新的整数数据与非频繁更新的字符串数据切分
    - 范例 user 表 ，个人简介，地址，QQ 号，联系方式，头像 这些字段为字符串类型，更新请求少； 最后登录时间，在线时常，访问次数，信件数这些字段为整数型字段，更新频繁，可以将后面这些更新频繁的字段独立拆出一张数据表，表内容变少，索引结构变少，读写请求变快。
  - 横向切表
    - 等分切表，如哈希切表或其他基于对某数字取余的切表。等分切表的优点是负载很方便的分布到不同服务器；缺点是当容量继续增加时无法方便的扩容，需要重新进行数据的切分或转表。而且一些关键主键不易处理。
    - 递增切表，比如每 1kw 用户开一个新表，优点是可以适应数据的自增趋势；缺点是往往新数据负载高，压力分配不平均。
    - 日期切表，适用于日志记录式数据，优缺点等同于递增切表。
    - 个人倾向于递增切表，具体根据应用场景决定。
  - 热点数据分表
    - 将数据量较大的数据表中将读写频繁的数据抽取出来，形成热点数据表。通常一个庞大数据表经常被读写的内容往往具有一定的集中性，如果这些集中数据单独处理，就会极大减少整体系统的负载。
    - 热点数据表与旧有数据关系
      - 可以是一张冗余表，即该表数据丢失不会妨碍使用，因源数据仍存在于旧有结构中。优点是安全性高，维护方便，缺点是写压力不能分担，仍需要同步写回原系统。
      - 可以是非冗余表，即热点数据的内容原有结构不再保存，优点是读写效率全部优化；缺点是当热点数据发生变化时，维护量较大。
      - 具体方案选择需要根据读写比例决定，在读频率远高于写频率情况下，优先考虑冗余表方案。
    - 热点数据表可以用单独的优化的硬件存储，比如昂贵的闪存卡或大内存系统。
    - 热点数据表的重要指标
      - 热点数据的定义需要根据业务模式自行制定策略，常见策略为，按照最新的操作时间；按照内容丰富度等等。
      - 数据规模，比如从 1000 万条数据，抽取出 100 万条热点数据。
      - 热点命中率，比如查询 10 次，多少次命中在热点数据内。
      - 理论上，数据规模越小，热点命中率越高，说明效果越好。需要根据业务自行评估。
    - 热点数据表的动态维护
      - 加载热点数据方案选择
        - 定时从旧有数据结构中按照新的策略获取
        - 在从旧有数据结构读取时动态加载到热点数据
      - 剔除热点数据方案选择
        - 基于特定策略，定时将热点数据中访问频次较少的数据剔除

        - 如热点数据是冗余表，则直接删除即可，如不是冗余表，需要回写给旧有数据结构。
    - 通常，热点数据往往是基于缓存或者 key-value 方案冗余存储，所以这里提到的热点数据表，其实更多是理解思路，用到的场合可能并不多….

反范式设计（冗余结构设计）

- 反范式设计的概念
    - 无外键，无连表查询。
    - 便于分布式设计，允许适度冗余，为了容量扩展允许适度开销。
    - 基于业务自由优化，基于 i/o 或查询设计，无须遵循范式结构设计。
- 冗余结构设计所面临的典型场景
    - 原有展现程序涉及多个表的查询，希望精简查询程序
    - 数据表拆分往往基于主键，而原有数据表往往存在非基于主键的关键查询，无法在分表结构中完成。
    - 存在较多数据统计需求（count, sum 等），效率低下。
- 冗余设计方案
    - 基于展现的冗余设计
        - 为了简化展现程序，在一些数据表中往往存在冗余字段
        - 举例，信息表 message，存在字段 fromuid,touid,msg,sendtime 四个字段，其中 touid+sendtime 是复合索引。存在查询为 select * from message where touid=$uid order by sendtime desc limit 0,30;
        - 展示程序需要显示发送者姓名，此时通常会在 message 表中增加字段 fromusername，甚至有的会增加 fromusersex，从而无需连表查询直接输出信息的发送者姓名和性别。这就是一种简单的，为了避免连表查询而使用的冗余字段设计。
    - 基于查询的冗余设计
        - 涉及分表操作后，一些常见的索引查询可能需要跨表，带来不必要的麻烦。确认查询请求远大于写入请求时，应设置便于查询项的冗余表。
        - 冗余表要点
            - 数据一致性，简单说，同增，同删，同更新。
            - 可以做全冗余，或者只做主键关联的冗余，比如通过用户名查询 uid，再基于 uid 查询源表。
        - 实战范例 1
            - 用户分表，将用户库分成若干数据表
            - 基于用户名的查询和基于 uid 的查询都是高并发请求。
            - 用户分表基于 uid 分成数据表，同时基于用户名做对应冗余表。
            - 如果允许多方式登陆，可以有如下设计方法
                - uid,passwd,用户信息等等，主数据表，基于 uid 分表
                - ukey,ukeytype,uid 基于 ukey 分表，便于用户登陆的查询。分解成如下两个 SQL。
                    - select uid from ulist_key_13 where ukey=’$username’ and ukeytype=‘login’;
                    - select * from ulist_uid_23 where uid=$uid and

passwd=’$passwd’;

- ukeytype 定义用户的登陆依据，比如用户名，手机号，邮件地址，网站昵称等。 Ukey+ukeytype 必须唯一。
- 此种方式需要登陆密码统一，对于第三方 connect 接入模式，可以通过引申额外字段完成。

◆ 实战范例 2：用户游戏积分排名

- 表结构 uid,gameid,score 参见前文实时积分排行。表内容巨大，需要拆表。
- 需求 1：基于游戏 id 查询积分排行
- 需求 2：基于用户 id 查询游戏积分记录
- 解决方案：建立完全相同的两套表结构，其一以 uid 为拆表主键，其二以 gameid 为拆表主键，用户提交积分时，向两个数据结构同时提交。

◆ 实战范例 3：全冗余查询结构

- 主信息表仅包括 主键及备注 memo 字段（text 类型），只支持主键查询，可以基于主键拆表。所以需要展现和存储的内容均在 memo 字段重体现。
- 对每一个查询条件，建立查询冗余表，以查询条件字段为主键，以主信息表主键 id 为内容。
- 日常查询只基于查询冗余表，然后通过 in 的方式从主信息表获得内容。
- 优点是结构扩展非常方便，只需要扩展新的查询信息表即可，核心思路是，只有查询才需要独立的索引结构，展现无需独立字段。
- 缺点是只适合于相对固定的查询架构，对于更加灵活的组合查询束手无策。

■ 基于统计的冗余结构

◆ 为了减少会涉及大规模影响结果集的表数据操作，比如 count，sum 操作。应将一些统计类数据通过冗余数据结构保存。

◆ 冗余数据结构可能以字段方式存在，也可能以独立数据表结构存在，但是都应能通过源数据表恢复。

◆ 实战范例：

- 论坛板块的发帖量，回帖量，每日新增数据等。
- 网站每日新增用户数等。
- 参见 Discuz 论坛系统数据结构，有较多相关结构。
- 参见前文分段积分结构，是典型用于统计的冗余结构。
- 后台可以通过源数据表更新该数字。
- Redis 的 Zset 类型可以理解为存在一种冗余统计结构。

■ 历史数据表

◆ 历史数据表对应于热点数据表，将需求较少又不能丢弃的数据存入，仅在少数情况下被访问。

主从架构

- 基本认识
  - 读写分离对负载的减轻远远不如分库分表来的直接。
  - 写压力会传递给从表，只读从库一样有写压力，一样会产生读写锁！
  - 一主多从结构下，主库是单点隐患，很难解决（如主库当机，从库可以响应读写，

但是无法自动担当主库的分发功能）
    - 主从延迟也是重大问题。一旦有较大写入问题，如表结构更新，主从会产生巨大延迟。
- 应用场景
    - 在线热备
    - 异地分布
        - 写分布，读统一。
        - 仍然困难重重，受限于网络环境问题巨多！
    - 自动障碍转移
        - 主崩溃，从自动接管
    - 个人建议，负载均衡主要使用分库方案，主从主要用于热备和障碍转移。
- 潜在优化点
    - 为了减少写压力，有些人建议主不建索引提升 i/o 性能，从建立索引满足查询要求。个人认为这样维护较为麻烦。而且从本身会继承主的 i/o 压力，因此优化价值有限。该思路特此分享，不做推荐。

故障转移处理

- 要点
    - 程序与数据库的连接，基于虚地址而非真实 ip，由负载均衡系统监控。
    - 保持主从结构的简单化，否则很难做到故障点摘除。
- 思考方式
    - 遍历对服务器集群的任何一台服务器，前端 web，中间件，监控，缓存，db 等等，假设该服务器出现故障，系统是否会出现异常？用户访问是否会出现异常。
    - 目标：任意一台服务器崩溃，负载和数据操作均会很短时间内自动转移到其他服务器，不会影响业务的正常进行。不会造成恶性的数据丢失。（哪些是可以丢失的，哪些是不能丢失的）

缓存方案

缓存结合数据库的读取

- Memcached 是最常用的缓存系统
- Mysql 最新版本已经开始支持 memcache 插件，但据牛人分析，尚不成熟，暂不推荐。
- 数据读取
    - 并不是所有数据都适合被缓存，也并不是进入了缓存就意味着效率提升。
    - 命中率是第一要评估的数据。
    - 如何评估进入缓存的数据规模，以及命中率优化，是非常需要细心分析的。
        - 实景分析： 前端请求先连接缓存，缓存未命中连接数据库，进行查询，未命中状态比单纯连接数据库查询多了一次连接和查询的操作；如果缓存命中率很低，则这个额外的操作非但不能提高查询效率，反而为系统带来了额外的负载和复杂性，得不偿失。

- 相关评估类似于热点数据表的介绍。
- 善于利用内存，请注意数据存储的格式及压缩算法。

- Key-value 方案繁多，本培训文档暂不展开。

缓存结合数据库的写入

- 利用缓存不但可以减少数据读取请求，还可以减少数据库写入 i/o 压力
- 缓存实时更新，数据库异步更新
  - 缓存实时更新数据，并将更新记录写入队列
  - 可以使用类似 mq 的队列产品，自行建立队列请注意使用 increment 来维持队列序号。
  - 不建议使用 get 后处理数据再 set 的方式维护队列
    - 测试范例：
      - 范例 1

        $var=Memcache_get($memcon,”var”);

        $var++;

        memcache_set($memcon,”var”,$var);

        这样一个脚本，使用 apache ab 去跑，100 个并发，跑 10000 次，然后输出缓存存取的数据，很遗憾，并不是 1000，而是 5000 多，6000 多这样的数字，中间的数字全在 get & set 的过程中丢掉了。

        原因，读写间隔中其他并发写入，导致数据丢失。
      - 范例 2

        用 memcache_increment 来做这个操作，同样跑测试

        会得到完整的 10000，一条数据不会丢。
    - 结论： 用 increment 存储队列编号，用标记+编号作为 key 存储队列内容。
  - 后台基于缓存队列读取更新数据并更新数据库
    - 基于队列读取后可以合并更新
    - 更新合并率是重要指标
      - 实战范例：

      某论坛热门贴，前端不断有 views=views+1 数据更新请求。

      缓存实时更新该状态

      后台任务对数据库做异步更新时，假设执行周期是 5 分钟，那么五分钟可能会接收到这样的请求多达数十次乃至数百次，合并更新后只执行一次 update 即可。

      类似操作还包括游戏打怪，生命和经验的变化；个人主页访问次数的变化等。
  - 异步更新风险
    - 前后端同时写，可能导致覆盖风险。
      - 使用后端异步更新，则前端应用程序就不要写数据库，否则可能造成写入冲突。一种兼容的解决方案是，前端和后端不要写相同的字段。
      - 实战范例：

        用户在线上时，后台异步更新用户状态。

        管理员后台屏蔽用户是直接更新数据库。

        结果管理员屏蔽某用户操作完成后，因该用户在线有操作，后台异步更新程序再次基于缓存更新用户状态，用户状态被复活，屏蔽失效。

- 缓存数据丢失或服务崩溃可能导致数据丢失风险。
  - 如缓存中间出现故障，则缓存队列数据不会回写到数据库，而用户会认为已经完成，此时会带来比较明显的用户体验问题。
  - 一个不彻底的解决方案是，确保高安全性，高重要性数据实时数据更新，而低安全性数据通过缓存异步回写方式完成。此外，使用相对数值操作而不是绝对数值操作更安全。
    - 范例：支付信息，道具的购买与获得，一旦丢失会对用户造成极大的伤害。而经验值，访问数字，如果只丢失了很少时间的内容，用户还是可以容忍的。
    - 范例：如果使用 Views=Views+…的操作，一旦出现数据格式错误，从 binlog 中反推是可以进行数据还原，但是如果使用 Views=特定值的操作，一旦缓存中数据有错误，则直接被赋予了一个错误数据，无法回溯！
- 异步更新如出现队列阻塞可能导致数据丢失风险。
  - 异步更新通常是使用缓存队列后，在后台由 cron 或其他守护进程写入数据库。
  - 如果队列生成的速度>后台更新写入数据库的速度，就会产生阻塞，导致数据越累计越多，数据库响应迟缓，而缓存队列无法迅速执行，导致溢出或者过期失效。
- 建议使用内存队列产品而不使用 memcache 来进行缓存异步更新。

总结

- 第一步，完成数据库查询的优化，需要理解索引结构，才能学会判断影响结果集。而影响结果集对查询效率线性相关，掌握这一点，编写数据查询语句就很容易判断系统开销，了解业务压力趋势。
- 第二步，在 SQL 语句已经足够优化的基础上，学会对数据库整体状况的分析，能够对异常和负载的波动有正确的认识和解读；能够对系统资源的分配和瓶颈有正确的认识。
- 学会通过监控和数据来进行系统的评估和优化方案设计，杜绝拍脑袋，学会抓大放小，把握要点的处理方法。
- 第三步，在彻底掌握数据库语句优化和运维优化的基础上，学会分布式架构设计，掌握复杂，大容量数据库系统的搭建方法。
- 最后，分享一句话，学会把问题简单化，正如 Caoz 常说的，你如果认为这个问题很复杂，你一定想错了。
- 感谢您的阅读，如对您有帮助，请在百度文库给本文五分好评，并推荐给您的朋友，多谢。